# Woche 1

In Woche 1 fand kein Unterricht statt.

# Woche 2

* Unterschiedliche Compiler
  + JIT – Just-in-Time (API Level 8)
    - Just-in-time-Kompilierung (JIT-Kompilierung) ist ein Verfahren, um Programme zur Laufzeit in Maschinencode zu übersetzen.
    - Ziel: Die Ausführungsgeschwindigkeit gegenüber einem Interpreter zu steigern.
    - JIT-Compiler kommen meist im Rahmen einer virtuellen Maschine zum Einsatz, wo plattform-unabhängiger Bytecode ausgeführt werden soll.
  + AOT – Ahead-of-Time (API Level 21)
    - Ein Ahead-of-time-Compiler (AOT-Compiler) ist ein Compiler der Android Runtime (ART), der im Gegensatz zu Just-in-time-Compilern (JIT-Compiler) Programmcode bereits bei der Installation des Programms übersetzt.
    - Vorteil: Dieser Code kann zur Laufzeit wesentlich schneller ausgeführt werden, da die Übersetzung bereits durchgeführt wurde.
    - Nachteil: Dieser Code ist nicht mehr plattformunabhängig
* Minimale und empfohlene Android Version
  + minSdkVersion: Legt den minimalen API Level fest, welcher von der Applikation unterstützt wird.
  + targetSdkVersion: Legt den API Level fest, auf den die Applikation ausgerichtet ist, d.h. optimal ausgeführt werden kann.
  + Definition im File "app/build.gradle" (Android-Studio), sonst File "AndroidManifest.xml"
  + Android Market nutzt diese Definitionen als Filter, um die Devices zu bestimmen auf denen eine entsprechende Applikation installiert werden kann 🡪 WICHTIG!
* Activity
  + Eine Activity repräsentiert ein sichtbares Benutzerinterface.
  + Die Activity enthält die Anwendungslogik.
  + Ein Activity besitzt eine View.
  + Die View beschreibt die Präsentation.
  + Die View wird normalerweise über ein XML-File beschrieben.
  + Im Lebenszyklus einer Applikation ist immer genau eine Activity aktiv.
  + Eine Applikation kann mehrere Activitys besitzen und zwischen diesen hin und her schalten.
* Application Workflow
  + Der Workflow setzt sich aus einer Sequenz von entsprechenden Activities zusammen
  + Verwaltung der Activities auf einem Stack
* Intent
  + Mittels der Klasse Intent werden Komponenten wie z.B. Activities gestartet.
  + Auch der Datenaustausch zwischen Activities kann durch die Klasse Intent erledigt werden.
  + Impliziter Intent: startActivity(new Intent("edu.intro.MyIntent"));
  + Expliziter Intent: startActivity(new Intent(Context context, Class<?> clazz))
* Application Lifecycle
  + Starten der Applikation
    - onCreate()
    - onStart()
    - onResume()
  + Wechseln auf den Desktop
    - onPause() 🡪 Fällt je nachdem weg
    - onStop()
  + Beenden via Taskmanager/Swipe
    - Witzigerweise nicht wie vermutet onDestroy() 🡪 Es wird nicht garantiert, dass onDestroy() je aufgerufen wird
  + Wechsel Portrait/Landscape Modus
    - onPause()
    - onStop()
    - onDestroy()
    - onCreate()
    - onStart()
    - onResume()
  + Telefonanruf
    - onPause() (Aber erst, wenn man den Anruf via Popup annimmt)
    - onResume() (Aber erst, wenn man wieder auflegt)
    - Obwohl man also den blauen Anrufbildschirm vor sich hat, wird nicht onStop() aufgerufen. Wechselt man via Task Manager zurück, passiert folgendes:
      * onStop() (Man sieht den Task Manager)
      * onRestart()
      * onStart()
      * onResume()
    - Nimmt man den Telefonanruf entgegen und sieht man den blauen Anrufbildschirm, kann man auch wieder zurückgehen und folgendes passiert:
      * onResume()
* MainActivty aus der 1. Übung

|  |
| --- |
| **package** ch.fhnw.edu.emoba.ueb1;  **import** android.app.Activity; **import** android.content.Intent; **import** android.os.Bundle; **import** android.util.Log; **import** android.view.View;  **public class** MainActivity **extends** Activity {  **private final static** String TAG = **"MainActivity"**;  **private final int** MY\_REQUEST\_CODE = 1;  **private final** String KEY\_MYVALUE = **"value"**;  **private int** value;   *// Create the view for the activty* @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  }   *// Reload existing cache values* @Override  **protected void** onRestoreInstanceState(Bundle savedInstanceState) {  **super**.onRestoreInstanceState(savedInstanceState);  **if** (savedInstanceState != **null**) {  value = savedInstanceState.getInt(KEY\_MYVALUE);  } **else** {  value = 0;  }  }   *// Update cache values* @Override  **protected void** onSaveInstanceState(Bundle outState) {  outState.putInt(KEY\_MYVALUE, value);  **super**.onSaveInstanceState(outState);  }   *// Provide a hook for activtiy results* @Override  **protected void** onActivityResult(**int** requestCode, **int** resultCode, Intent data) {  **if** (requestCode == MY\_REQUEST\_CODE) {  **if** (data != **null**) {  **int** counter = data.getIntExtra(**"counter"**, 0);  Log.d(TAG, **"Received for requestCode="** + requestCode + **" the resultCode="** + resultCode + **" with counter="** + counter);  } **else** {  Log.d(TAG, **"Received for requestCode="** + requestCode + **" the resultCode="** + resultCode);  }  }  }   *// Provide a button click hoock* **public void** onButtonClick(View v) {  Intent intent = **new** Intent(**"ch.fhnw.edu.helloworld.HELLOWORLD"**);  *// Intent intent = new Intent(this, CounterActivity.class);   // startActivity(intent);* startActivityForResult(intent, MY\_REQUEST\_CODE);  } } |

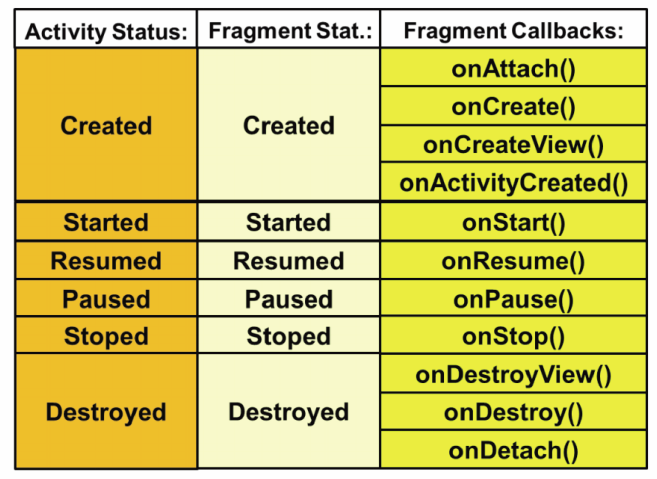
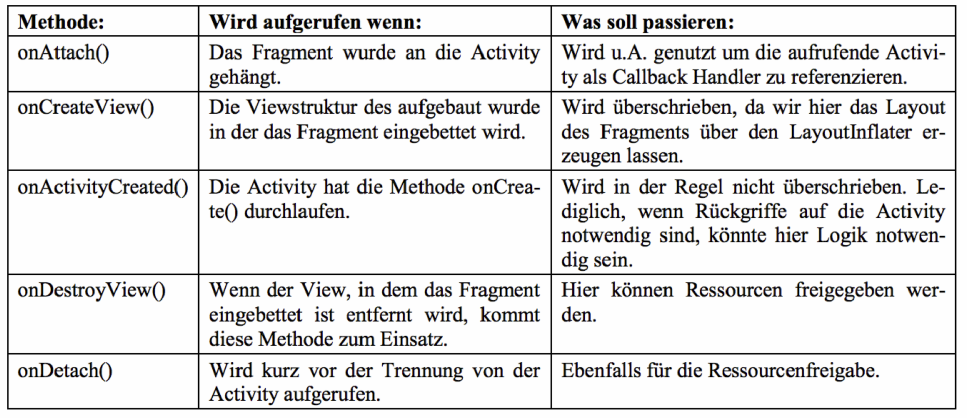
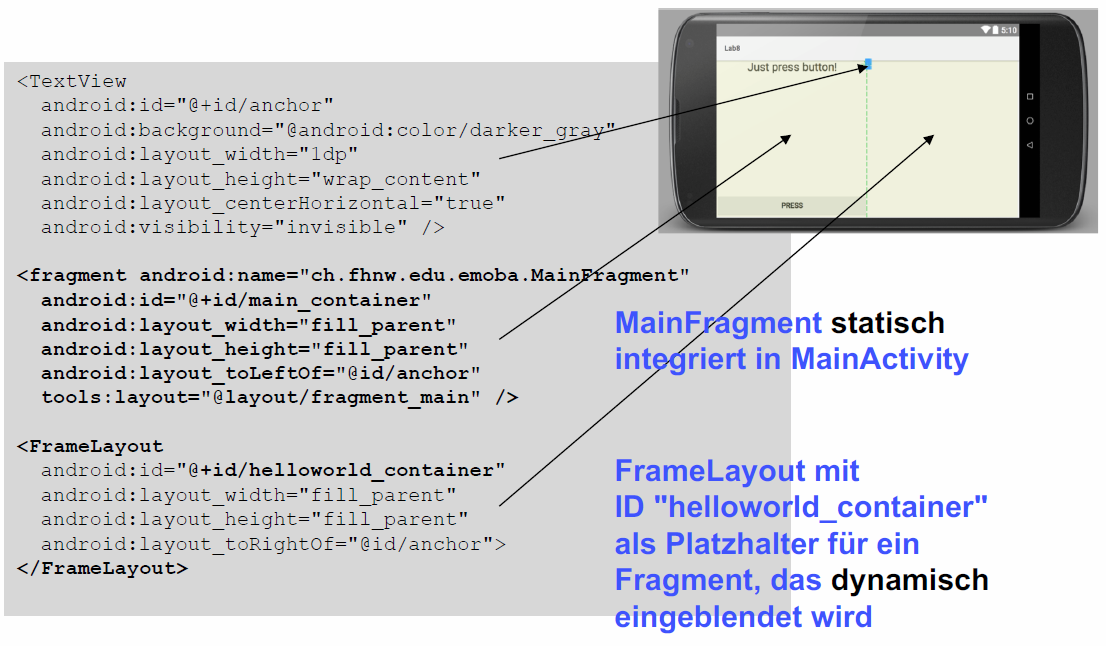
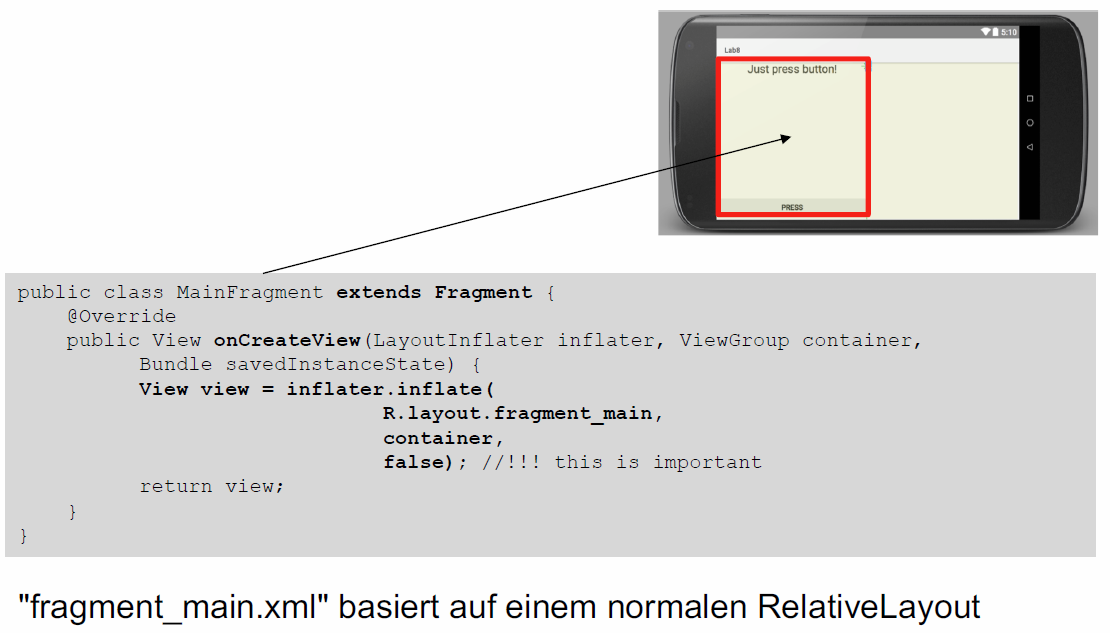
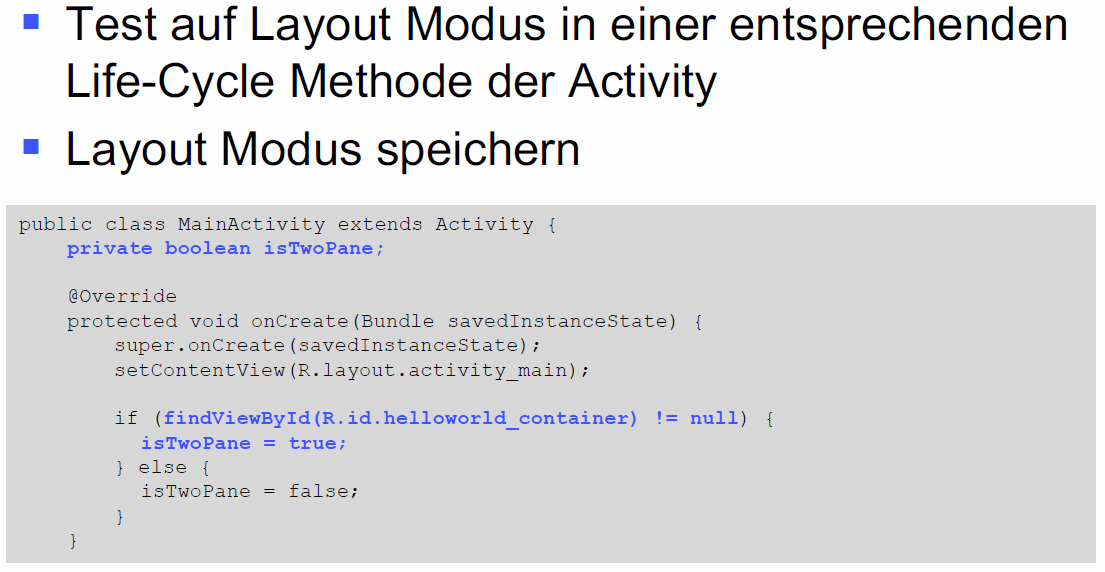
* CounterActivity aus der 1.Übung

|  |
| --- |
| **package** ch.fhnw.edu.emoba.ueb1;  **import** android.app.Activity; **import** android.content.Intent; **import** android.os.Bundle; **import** android.util.Log; **import** android.widget.TextView;  **public class** CounterActivity **extends** Activity {  **private final static** String TAG = **"CounterActivity"**;  **private final** String STATE\_COUNTER = **"counter"**;  **private int** counter;   *// Create the view and load the cached value* @Override  **public void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.counter);   TextView counterView = (TextView) findViewById(R.id.counterView);  counterView.setText(String.valueOf(counter));  *// the intent must be set before onPause(), onStop(), ...* Intent responseIntent = **new** Intent().putExtra(**"counter"**, counter);  setResult(RESULT\_OK, responseIntent);  }   */\*\*  \* Task 4  \*/* @Override  **protected void** onPause() {  *//finish();* **super**.onPause();  }   *// Reload existing cache values* @Override  **protected void** onRestoreInstanceState(Bundle savedInstanceState) {  **super**.onRestoreInstanceState(savedInstanceState);  **if** (savedInstanceState != **null**) {  counter = savedInstanceState.getInt(STATE\_COUNTER);  } **else** {  counter = 0;  }  *// Update view with actual counter value* TextView counterView = (TextView) findViewById(R.id.counterView);  counterView.setText(String.valueOf(counter));   *// Update response with actual counter value* Intent responseIntent = **new** Intent().putExtra(**"counter"**, counter);  setResult(RESULT\_OK, responseIntent);  }   *// Update cache values* @Override  **protected void** onSaveInstanceState(Bundle outState) {  outState.putInt(STATE\_COUNTER, ++counter);  **super**.onSaveInstanceState(outState);  } } |

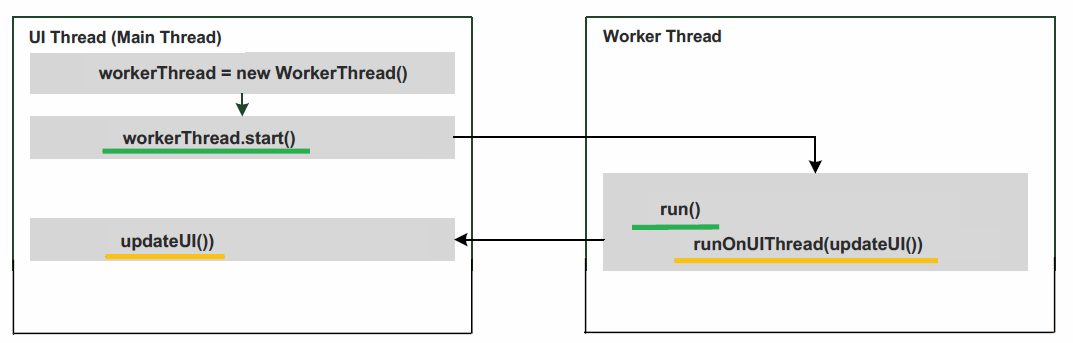
# Woche 3

In Woche 3 fand kein Unterricht statt.

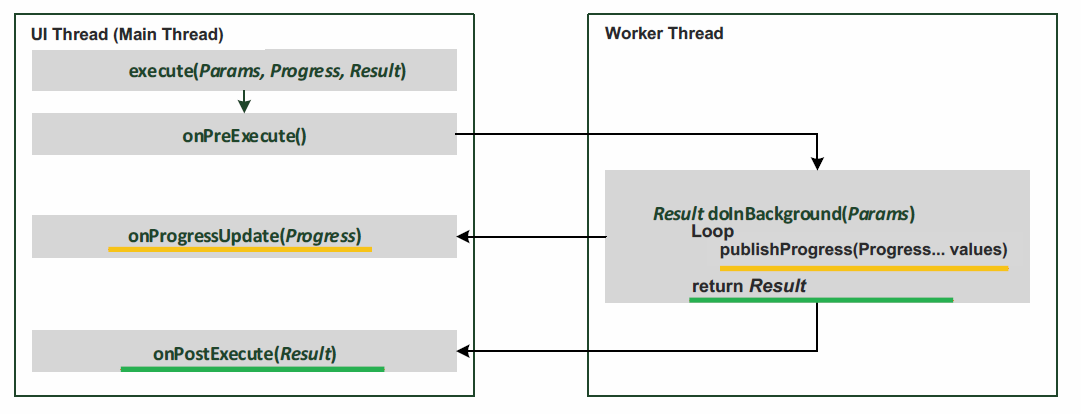
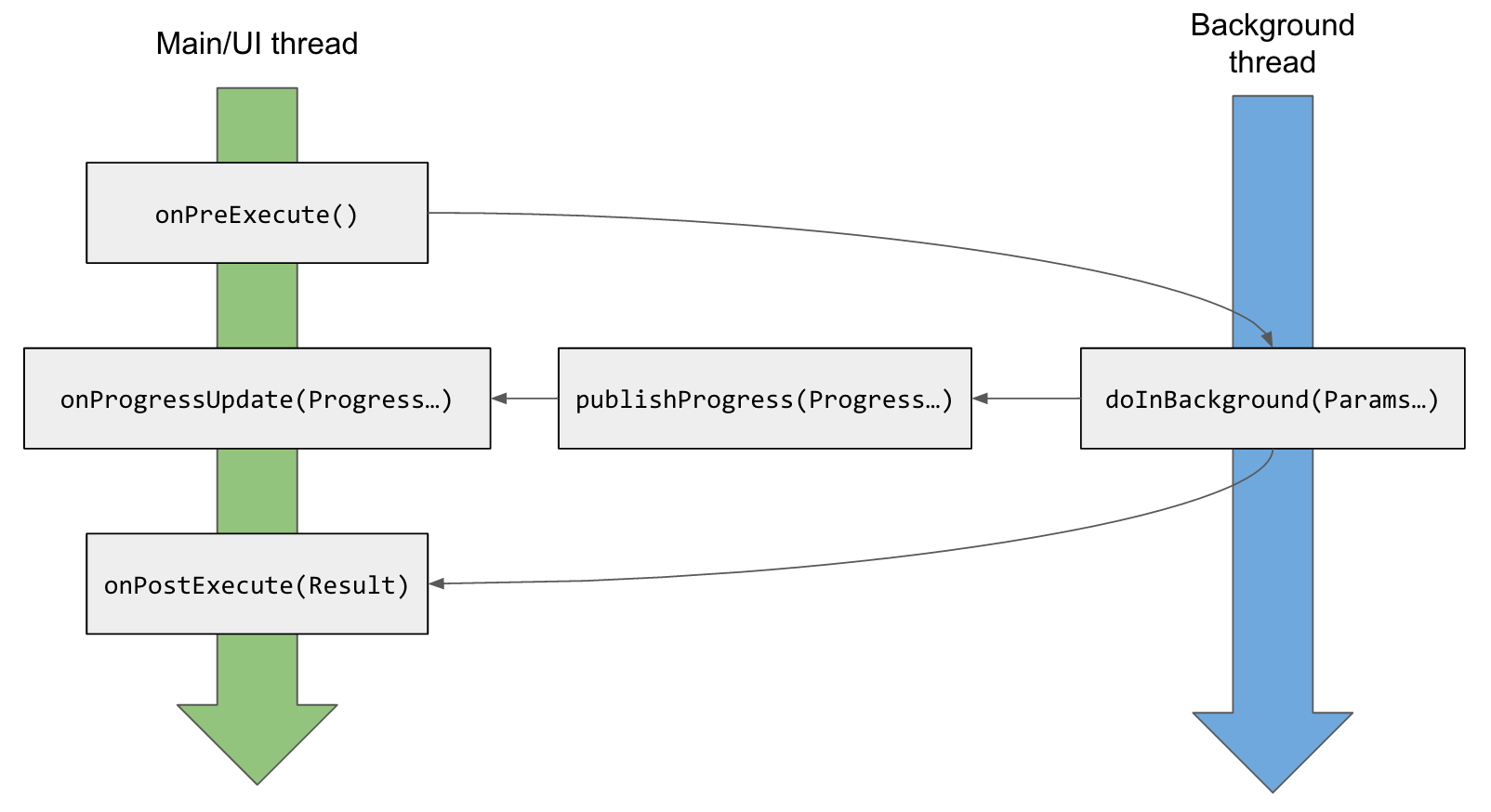
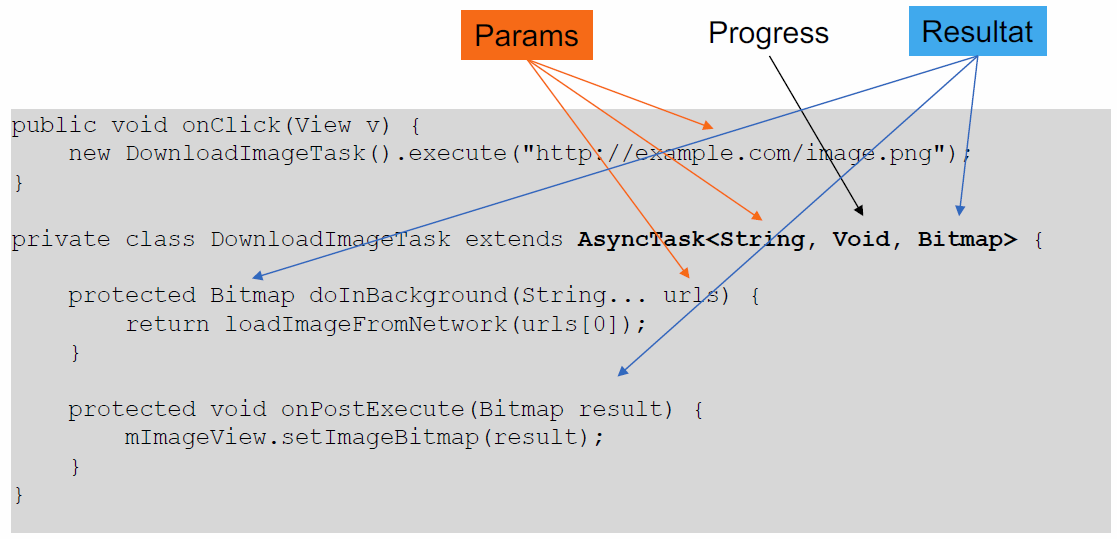
# Woche 4

* GUI Performance
  + In onCreate() wird normalerweise das View über setContentView gesetzt
  + Im setContentView() wird die View aus dem XML File aufgebaut (welche aber zu einer Java/Bytecodedatei umkompiliert worden ist)
  + Der GUI Aufbau kann zeitintensiv sein, falls das Layout komplex ist
  + Für ein gutes Benutzererlebnis muss die Startzeit kurz gehalten werden
    - Ansatz 1:
      * Keine tiefen Verschachtelungen – Layout merge (Fasst zusammengebaute Views mit einer und nur einem Layout zusammen 🡪 Schnell aber nur ein gemeinsames Layout) vs. Include (Kopiert einfach alles zusammen 🡪 Langsam aber flexibel)
      * RelativeLayout (Arbeitet relativ zum Parent) anstelle von [genesteten!] LinearLayout (Fügt alle schön langsam nacheinander aneinander)
      * ConstraintLayout anstelle von RelativeLayout, da dieses die Viewtiefe verkleinert und Nesting verhindert
    - Ansatz 2: Lazy Initialization mit ViewStub und Verwendung des Factory Method Pattern
* Herausforderung Device Fragmentierung (Unterschiedliche Versionen und Bildschirmgrössen)
* Lösungskonzept Alternative Resources
  + Unter Alternative Resources versteht man die Möglichkeit, verschiedenen Ressourcensätze bereitstellen zu können, so dass das System aufgrund seiner Ausstattung den passenden Ressourcensatz wählen kann
  + Ressourcen können unterteilt werden in
    - Internationalisierung
    - Screen Resolution (Pixel Density)
    - Screen Orientation (Portrait vs. Landscape)
  + Ressourcen wie Bilder oder Strings sollten auf jeden Fall externalisiert werden, so dass eine Unabhängigkeit zwischen Code und Ressourcen entsteht, was die Wartbarkeit und Erweiterbarkeit stark vereinfacht, zum Beispiel Internationalisierung
  + Das /res Verzeichnis verwaltet alle Ressourcen und muss einer bestimmten Konvention folgen:
    - «layout» für das User Layout (Portrait vs. Landscape)
    - «drawable» für Bitmap Dateien (Bilder)
    - «mipmap» für Launcher Icons
    - Und viele andere
  + Klasse R:Bei der Kompilation der Applikazion werden alle Resource-IDs einer Ressource unterhalb des «/res» Directories in die Java Klasse R integriert
  + Auf Ressourcen zugreifen
    - Aus Java über die R Klasse: R.string.hello
    - Aus XML über eine spezialisierte XML Syntax: @string/hello
* Lösungskonzept Component Fragments
  + Ausgangslage: Mit den Tablets stiess das Konzept «Activity» an seine Grenzen, da eine App sehr unterschiedliche Bildschirmgrössen unterstützen musste. Deshalb wurde das Konzept «Fragment» entwickelt und mit Android 3.0 (API 11) eingeführt
  + Komponente «Fragment»: Fragments sind Teile einer Activity, welche unabhängig voneinander und unabhängig vom Layout geladen und mit Funktionalität bestückt werden können
  + 
  + 
  + 
  + 
  + 
  + 

# Woche 5

* Main-Thread Konzept
  + Im Main Thread (oder UI Thread) wird die Applikation ausgeführt
    - Key Events empfangen (View.onKeyDown())
    - Views zeichnen (View.onDraw())
    - Life Cycle Events verarbeiten (Activity.onCreate())
  + Der Main Thread erhält alle Notifications über das was auf dem Device abläuft, ausgelöst vom System und/pder Benutzer
  + Jede Applikation hat immer nur einen Main Thread. Alle Events werden deshalb sequentiell abgearbeitet
    - Die Bearbeitung jedes Events muss schnell erfolgen
    - Multithreading wird notwendig
* Multithreading mit Java Thread
  + Updates des UIs aus einem Worker Thread?
    - Wichtig: Das UI kann nur aus dem Main Thread geändert werden
  + Methode: runOnUiThread(Runnable action): Führt die Runnable Aktion, sofern auf dem Main Thread, gleich direkt auf oder legt sie via post(Runnable action) auf die azuarbeitende Queue
  + Methode post(Runnable action): Queue welche vom Main Thread abgearbeitet wird
  + 
  + MainActivity.java aus dem 4. Arbeitsblatt

|  |
| --- |
| **package** ch.fhnw.edu.emoba.ab4;  **import** android.app.Activity; **import** android.graphics.Color; **import** android.os.Bundle; **import** android.view.View; **import** android.widget.TextView;  **import** java.util.concurrent.atomic.AtomicBoolean;  **public class** MainActivity **extends** Activity {   **private static final** String ***STATE\_RUNNING*** = **"task.running"**;   **private** Thread **colorThread**;   **private** AtomicBoolean **isCancelled**;   **private void** updateHelloWorldView(**int** red, **int** green, **int** blue) {  TextView textView = findViewById(R.id.***txtView***);  textView.setBackgroundColor(Color.*rgb*(red, green, blue));  }   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main***);   TextView textView = findViewById(R.id.***txtView***);  textView.setText(getResources().getString(R.string.***app\_name***));   *// Check if we have to restore and rerun the thread* **isCancelled** = **new** AtomicBoolean();  **if** (savedInstanceState != **null** && savedInstanceState.containsKey(***STATE\_RUNNING***)) {  *// Restart* **colorThread** = **new** ColorThread();  **colorThread**.start();  } **else** {  *// Idle* **isCancelled**.set(**true**);  }  }   @Override  **protected void** onSaveInstanceState(Bundle outState) {  *// Check if the thread is running and we have to rerun it later on* **super**.onSaveInstanceState(outState);  **if** (**colorThread** != **null**) {  outState.putBoolean(***STATE\_RUNNING***, **true**);  **isCancelled**.set(**true**);  }  }   **public void** start(View view) {  *// Wipe the cancel flag and fire up the thread* **if** (**colorThread** == **null**) {  **isCancelled**.set(**false**);  **colorThread** = **new** ColorThread();  **colorThread**.start();  }  }   **public void** stop(View view) {  *// Set the cancel flag and dereference the thread* **if** (**colorThread** != **null**) {  **isCancelled**.set(**true**);  **colorThread** = **null**;  }  }   **class** ColorThread **extends** Thread {   **private boolean errorFlag**;   @Override  **public void** run() {  **try** {  **while** (!**errorFlag** && !**isCancelled**.get()) {  **final int** red = (**int**) Math.*round*(Math.*random*() \* 255);  **final int** green = (**int**) Math.*round*(Math.*random*() \* 255);  **final int** blue = (**int**) Math.*round*(Math.*random*() \* 255);  runOnUiThread(**new** Runnable() {   @Override  **public void** run() {  updateHelloWorldView(red, green, blue);  }  });  Thread.*sleep*(1000);  }  } **catch** (InterruptedException exception) {  **errorFlag** = **true**;  }  }  } } |

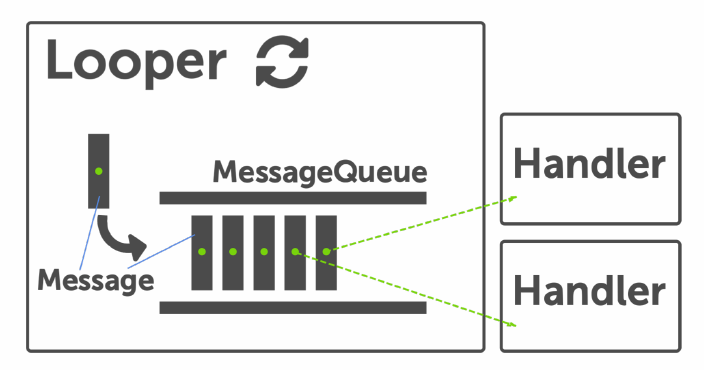
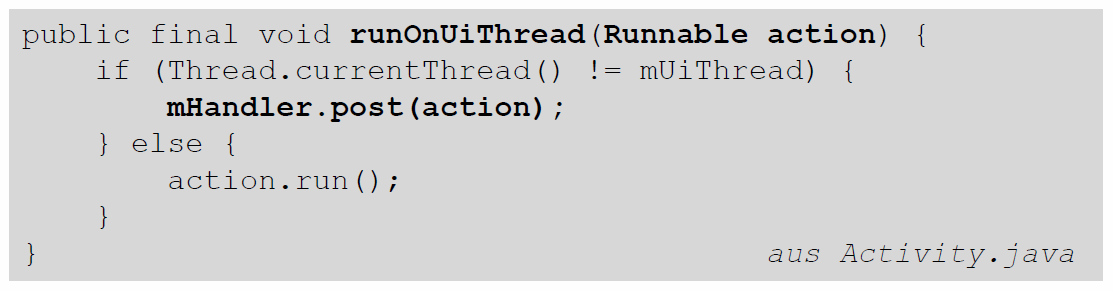
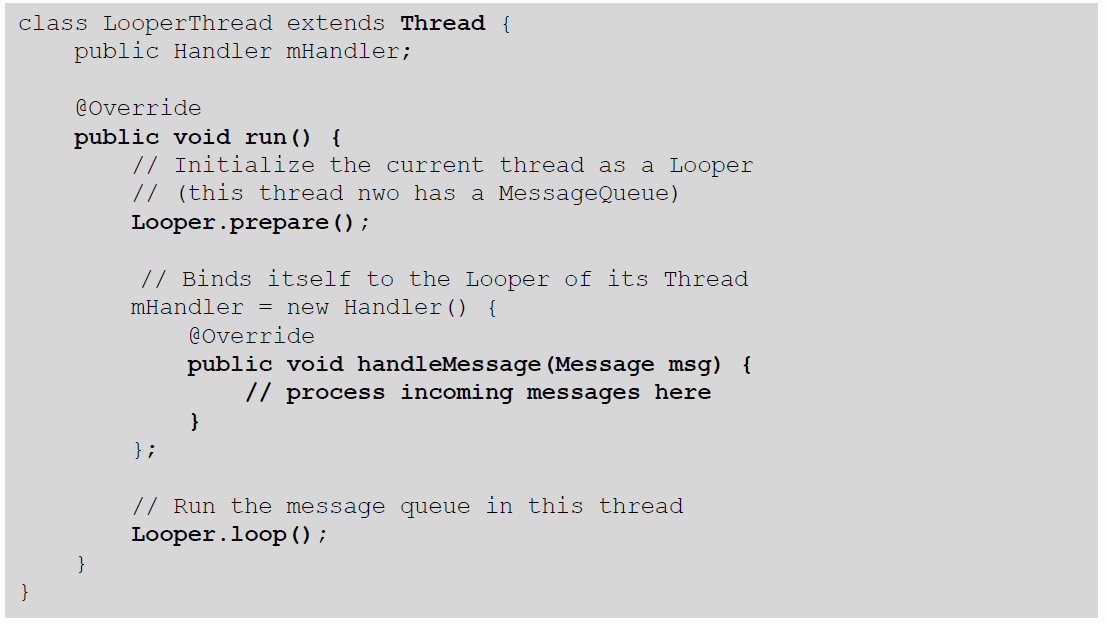
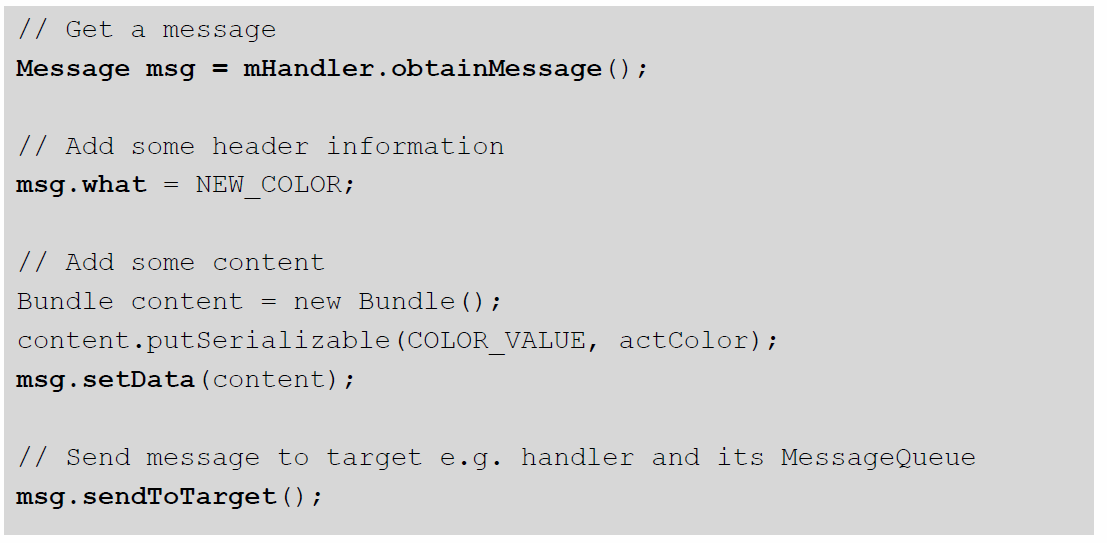
* Multi-Threading mit Async Task
  + Folgendes Pattern vereinfachen
    - Ein Event wird im UI Thread empfangen
    - Die entsprechende Anwendungslogik muss in einem non-UI Thread (=Worker Thread) ausgeführt werden
    - Das UI muss aufgrund des Resultats aktualisiert werden
  + Die Klasse AsyncTask ist als eine Helper-Klasse konzipiert, die kurze asynchrone Berechnungen auf einem Worker Thread Ausführt
  + 
  + 
  + 

|  |
| --- |
| **private class** DownloadFilesTask **extends** AsyncTask<URL, Integer, Long> {    **protected** Long doInBackground(URL... urls) {  **int** count = urls.**length**;  **long** totalSize = 0;  **for** (**int** i = 0; i < count; i++) {  totalSize += Downloader.downloadFile(urls[i]);  publishProgress((**int**) ((i / (**float**) count) \* 100));  *// Escape early if cancel() is called* **if** (isCancelled()) **break**;  }  **return** totalSize;  }   **protected void** onProgressUpdate(Integer... progress) {  setProgressPercent(progress[0]);  }   **protected void** onPostExecute(Long result) {  showDialog(**"Downloaded "** + result + **" bytes"**);  } } |

* + MainActivity.java aus dem 5. Arbeitsblatt

|  |
| --- |
| **package** ch.fhnw.edu.emoba.ab5;  **import** android.app.Activity; **import** android.graphics.Color; **import** android.os.AsyncTask;  **import** android.os.Bundle; **import** android.view.View; **import** android.widget.TextView;  **public class** MainActivity **extends** Activity {   **private static final** String ***STATE\_RUNNING*** = **"task.running"**;   **private** AsyncTask<Void, RgbColor, Void> **colorAsyncTask**;   **private void** updateHelloWorldView(RgbColor rgbColor) {  TextView textView = (TextView) findViewById(R.id.***txtView***);  textView.setBackgroundColor(Color.*rgb*(rgbColor.**red**, rgbColor.**green**, rgbColor.**blue**));  }   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main***);   TextView textView = (TextView) findViewById(R.id.***txtView***);  textView.setText(getResources().getString(R.string.***app\_name***));   **if** (savedInstanceState != **null** && savedInstanceState.containsKey(***STATE\_RUNNING***)) {  **colorAsyncTask** = **new** ColorAsyncTask();  **colorAsyncTask**.execute();  }  }   @Override  **protected void** onSaveInstanceState(Bundle outState) {  **super**.onSaveInstanceState(outState);  **if** (**colorAsyncTask** != **null**) {  outState.putBoolean(***STATE\_RUNNING***, **true**);  **colorAsyncTask**.cancel(**true**);  }  }   **public void** start(View view) {  **if** (**colorAsyncTask** == **null**) {  **colorAsyncTask** = **new** ColorAsyncTask();  **colorAsyncTask**.execute();  }  }   **public void** stop(View view) {  **if** (**colorAsyncTask** != **null**) {  **colorAsyncTask**.cancel(**true**);  **colorAsyncTask** = **null**;  }  }   **public class** ColorAsyncTask **extends** AsyncTask<Void, RgbColor, Void> {   **private boolean errorFlag**;   @Override  **protected void** onPreExecute() {  **errorFlag** = **false**;  }   @Override  **protected** Void doInBackground(Void[] voids) {  **try** {  **while** (!**errorFlag** && !isCancelled()) {  RgbColor rgbColor = **new** RgbColor();  rgbColor.**red** = (**int**) Math.*round*(Math.*random*() \* 255);  rgbColor.**green** = (**int**) Math.*round*(Math.*random*() \* 255);  rgbColor.**blue** = (**int**) Math.*round*(Math.*random*() \* 255);  publishProgress(rgbColor);  Thread.*sleep*(1000);  }  } **catch** (InterruptedException exception) {  **errorFlag** = **true**;  }  **return null**;  }   @Override  **protected void** onProgressUpdate(RgbColor[] values) {  updateHelloWorldView(values[0]);  }  }   **class** RgbColor {  **int red**;  **int green**;  **int blue**;  } } |

# Woche 6

* Linux Prozess
  + Beim Neustart einer Applikation startet das Android System einen neuen Prozess
  + Der Prozess erhält einen einzigen Thread (Main Thread)
  + Per Default werden alle Aktivitäten in diesem Main Thread ausgeführt
  + Jede Applikation wird in seinem eigenen Prozess ausgeführt und erhält eine eindeutige User-ID
  + Jeder Prozess erhält seinen eigenen Memory Space
  + Prozesse kommunizieren untereinander über IPC Konzepte wie Pipes oder Sockets
  + Die Laufzeit eines Prozesses wird über das Android System verwaltet, das heisst, ein Prozess wird erst beendet, falls die Applikation nicht mehr verwendet wird UND das System Memory für andere Applikationen bereitstellen muss
* Android Thread
  + Ein Thread ist die kleinste Ausführungseinheit, die vom Android System (Scheduler) unabhängig verwaltet wird
  + Alle Threads werden im gleichen Prozess ausgeführt
  + Threads teilen sich das Memory, haben aber einen eigenen Call Stack
  + Threads kommunizieren untereinander über Shared Objects oder Message Passing
* Main Thread
  + Jede Android Applikation hat einen einzigen Main Thread, der das User Interface koordiniert und kontrolliert
  + Main Thread wird auch UI Thread genannt
  + Alle Komponenten (Activity, Service, BroadcastReceiver etc.) einer Android Applikation werden im Main Thread instanziiert
  + Beispiel: Bei ienem Touch auf den Screen setzt der UI Thread den State des entsprechenden Widgets, generiert einen Event und postet einen “Invalidate Request” an die MessageQueue. Der UI Thread liest den Request und benachrichtigt das Widget sich neu zu zeichnen
* Multi-Threading
  + Worker Thread Varianten bisher:
    - Java Thread (siehe Lektion 3)
    - AsyncTask (siehe Lektion 3)
  + Kommunikation zwischen Main Thread und Worker Thread
    - Java Thread: runOnUiThread(), post()
    - AsyncTask: doInBackground() vs. onPreExecute(), onPostExecute()
* AsyncTask: Vor- und Nachteile
  + Vorteile
    - Sehr gut unterstützt (ab API Level 3, Version 1.5)
    - Sehr bekannt in der Android Entwickler Community
  + Nachteile
    - Lifecycle Verhalten: Obwohl eine Activity abschlossen werden könnte, wird die Methode doInBackground() solange ausgeführt bis diese Aufgabe abgeschlossen ist 🡪 AsyncTask muss explizit mit cancel(true) gestoppt werden, bevor die Activity beendet werden kann.
    - Memory Leaks: AsyncTask hält eine Referenz auf seine Activity, da seine Callback-Methoden sowohl im Worker-Thread wie auch im Main-Thread ausgeführt werden müssen à enge Koppelung zwischen AsyncTask und Activity. Falls die Activity beendet wird (z.B. Orientation Change) bleibt diese Referenz vorhanden, falls Worker-Thread nicht explizit gestoppt wird und die Activity kann nicht aus dem Memory entfernt werden.
    - Verlust des Resultates: Falls die Activity wieder erzeugt wird (z.B. Orientation Change) wird diese neue Referenz nicht an den AsyncTask weitergegeben à UI-Updates funktionieren nicht mehr.
* Lose Koppelung Main- & Worker-Thread
  + Enge Koppelung 🡪 AsyncTask: Lifecycle des Worker Threads entspricht der seiner Activity
    - Wichtig: explizites Stoppen des AsyncTask nicht vergessen 🡪 asyncTask.cancel(true)
  + Lose Koppelung 🡪 Handler: Thread Kommunikation ist möglich mittels Message Passing über die Klassen MessageQueue, Looper und Handler.
* Main Thread und «Message Passing»
  + Der Main- bzw. UI Thread besitzt immer folgende Objekte für eine sequentielle Abarbeitung der "Events":
    - MessageQueue: verwaltet Messages
    - Looper: hält Thread am Leben und liest aus der MessageQueue die nächste Message aus und ruft über die Message den assoziierten Handler auf
    - Handler: führt die Arbeit auf der entsprechenden Message aus
  + 
* runOnUiThread() Details
  + 
  + Jede Applikation/Activity hat einen einzigen Main Thread (UI Thread).
  + Der Main Thread hat einen Looper, eine MessageQueue und einen Handler (z.B. mHandler).
  + Über den Handler können Runnable-Objekte in die MessageQueue des Main Threads gelegt werden (z.B. post(...)), um so eigene Anweisungen im Main Thread ausführen zu können.
* Class Handler: API
  + "A Handler allows you to send Message and Runnable objectsand”
  + “A Handler allows you to process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue."
* Class Message: API
  + "Defines a message containing a description and arbitrary data object that can be sent to a Handler."
* Class Looper: API
  + "Class used to run a message loop for a thread. Threads by default do not have a message loop associated with them; to create one, call prepare() in the thread that is to run the loop, and then loop() to have it process messages until the loop is stopped.
  + Most interaction with a message loop is through the Handler class."
  + 
* Class HandlerThread: API
  + "Handy class for starting a new thread that has a looper. The looper can then be used to create handler classes. Note that start() must still be called."
  + 
  + 
* MainActivity.java aus dem 6. Arbeitsblatt

|  |
| --- |
| **package** ch.fhnw.edu.emoba.ab6;  **import** android.app.Activity; **import** android.os.Bundle; **import** android.os.Handler; **import** android.os.Message; **import** android.util.Log; **import** android.view.View; **import** android.widget.TextView;  **public class** MainActivity **extends** Activity {   **public static final** String ***TAG*** = **"MainActivity"**;   **public static int** *PING\_MESSAGE* = 0;   **private** TextView **helloWorldView**;   **private** WorkerThread **workerThread** = **null**;   **private void** updateHelloWorldView(String text) {  **helloWorldView**.setText(text);  }   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.***activity\_main***);  **helloWorldView** = findViewById(R.id.***txtView***);  }   @Override  **protected void** onResume() {  **super**.onResume();   Handler mainThreadHandler = **new** Handler(getMainLooper()) {  @Override  **public void** handleMessage(Message msg) {  *// what will be sent to the main thread? What and data.* **if** (msg.**what** == WorkerThread.***WRITE\_TEXT***) {  String message = msg.getData().getString(**"message"**);  updateHelloWorldView(message);  }  **if** (msg.**what** == WorkerThread.***PONG\_MESSAGE***) {  String message = msg.getData().getString(**"message"**);  updateHelloWorldView(message);  }  }  };   **workerThread** = **new** WorkerThread(**"worker"**, mainThreadHandler);  **workerThread**.start(); *// Thread wird gestartet.* Handler handler = **new** Handler(**workerThread**.getLooper());  handler.post(**new** Runnable() {   @Override  **public void** run() {  String name = Thread.*currentThread*().getName();  Log.*d*(name, **"Main Thread is ready"**);  }  });  }   *// gets executed when "text message" button gets clicked* **public void** showText(View view) {  **if** (**workerThread** != **null**) {  **workerThread**.showText();   }  }   **public void** showPingPong(View view) {  **if** (**workerThread** != **null**) {  Handler workerThreadHandler = **workerThread**.getWorkerThreadHandler();  **if** (workerThreadHandler != **null**) {  Message message = workerThreadHandler.obtainMessage();  message.**what** = *PING\_MESSAGE*;  Bundle bundle = **new** Bundle();  bundle.putString(**"message"**, **"PING"**);  message.setData(bundle);  workerThreadHandler.sendMessage(message);  }  }  }   @Override  **protected void** onPause() {  **super**.onPause();  *// see if a worker is started* **if** (**workerThread** != **null**) {  *// if so... quit it. Give resources free...* **workerThread**.quit();  **workerThread** = **null**;  Log.*d*(***TAG***, **"Shutting down"**);  }  } } |

* WorkerThread.java aus dem 6. Arbeitsblatt

|  |
| --- |
| **package** ch.fhnw.edu.emoba.ab6;  **import** android.os.Bundle; **import** android.os.Handler; **import** android.os.HandlerThread; **import** android.os.Message; **import** android.util.Log;  **public class** WorkerThread **extends** HandlerThread {   **public static final int *WRITE\_TEXT*** = 0;   **public static final int *PONG\_MESSAGE*** = 1;   **private** Handler **mainThreadHandler**;   **private** Handler **workerThreadHandler** = **null**;   **public** WorkerThread(String name, Handler handler) {  **super**(name);  **this**.**mainThreadHandler** = handler;  }   @Override  **protected void** onLooperPrepared() {  **super**.onLooperPrepared();   *//final Handler handler = new Handler(Looper.getMainLooper());* **final** Handler handler = **new** Handler(**mainThreadHandler**.getLooper());  handler.post(**new** Runnable() {   @Override  **public void** run() {  *// here we send something to the Main thread* String name = Thread.*currentThread*().getName();  Log.*d*(name, **"Worker Thread is ready"**);  }  });   *// this.workerThreadHandler = new Handler() {* **this**.**workerThreadHandler** = **new** Handler(getLooper()) {  @Override  **public void** handleMessage(Message msg) {  **if** (msg.**what** == MainActivity.*PING\_MESSAGE*) {  String message = msg.getData().getString(**"message"**);  **if** (**mainThreadHandler** != **null**) {  Message pong = **mainThreadHandler**.obtainMessage();  pong.**what** = ***PONG\_MESSAGE***;  Bundle bundle = **new** Bundle();  bundle.putString(**"message"**, message + **"/PONG"**);  pong.setData(bundle);  **mainThreadHandler**.sendMessage(pong);  }  }  }  };  }   **public void** showText() {  **if** (**mainThreadHandler** != **null**) {  String name = Thread.*currentThread*().getName();  Message message = **mainThreadHandler**.obtainMessage();  message.**what** = ***WRITE\_TEXT***;  Bundle content = **new** Bundle();  content.putString(**"message"**, **"Hello from WorkerThread: "** + name);  message.setData(content);  **mainThreadHandler**.sendMessage(message);  }  }   **public** Handler getWorkerThreadHandler() {  **return this**.**workerThreadHandler**;  } } |